VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PŘEKLADAČ JAZYKA VHDL PRO POTŘEBY FORMÁLNÍ VERIFIKACE

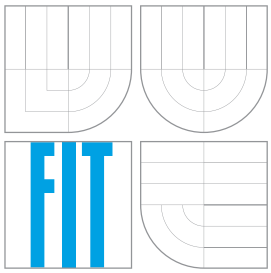
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JIŘÍ MATYÁŠ

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PŘEKLADAČ JAZYKA VHDL PRO POTŘEBY FORMÁLNÍ VERIFIKACE

A VHDL PARSER FOR FORMAL VERIFICATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JIŘÍ MATYÁŠ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. LUKÁŠ CHARVÁT

BRNO 2015

Abstrakt

Cílem této bakalářské práce je navrhnout a implementovat překladač, který umožňuje převod popisu hardware z jazyka VHDL do grafové reprezentace v jazyce VAM (Variable Assignment Language). Program je určen pro potřeby formální verifikace výzkumné skupiny VeriFIT Fakulty informačních technologií VUT Brno. Důvodem vypracování této práce je poskytnutí možnosti formálně verifikovat návrh hardware s využitím vysokoúrovňových návrhových jazyků, jakým je například jazyk VHDL.

Abstract

The principal goal of this bachelor thesis is to design and implement a parser of VHDL language into graph representation in VAM (Variable Assignment Language). The application is developed for formal verification purposes of VeriFIT research group of the Faculty of Information Technology, Brno University of Technology. The development of the compiler described in this thesis should provide the opportunity to use formal verification techniques to verify hardware designs described in high level design languages, such as VHDL.

Klíčová slova

VHDL překladač, Variable Assignment Model, formální verifikace, Icarus Verilog, data-flow graf, VVP mezikód

Keywords

VHDL parser, Variable Assignment Model, formal verification, Icarus Verilog, data-flow graph, VVP

Citace

Jiří Matyáš: Překladač jazyka VHDL
pro potřeby formální verifikace, bakalářská práce, Brno, FIT VUT v Brně, 2015

Překladač jazyka VHDL pro potřeby formální verifikace

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Lukáše Charváta.

.....
Jiří Matyáš
20. května 2015

Poděkování

Děkuji vedoucímu této bakalářské práce panu Ing. Lukáši Charvátovi za aktivní odbornou pomoc a podněty při řešení této práce.

Obsah

Kapitola 1

Úvod

Vývoj číslicových systémů je v současné době složitá a časově náročná činnost. Rostoucí složitost číslicových obvodů způsobuje větší riziko výskytu chyb v navrhovaném hardware. Tyto chyby se vývojáři nejčastěji snaží odhalit a eliminovat pomocí testování, které zabírá podstatnou část procesu návrhu a vývoje HW. Problémem je, že ani důkladné testování nemusí odhalit všechny chyby vyskytující se v obvodu. Proto se v dnešní době uplatňují také další přístupy pro ověření správnosti návrhu. Jedním z nich je formální verifikace, s jejíž pomocí je na rozdíl od testování možné potenciálně dokázat, že daný obvod je navržený správně. Díky odlišným principům oproti testování, na kterých je formální verifikace založena, je také často schopna odhalit nové typy chyb v návrhu. Program vzniklý v rámci této bakalářské práce by měl usnadnit použití verifikačních prostředků pro návrhy číslicových systémů popsané v jazycích pro popis hardware.

Cílem této práce je návrh a implementace překladače, jež převádí popis procesoru v jazyce VHDL nebo Verilog do grafové reprezentace toku dat v jazyce VAM. Jazyk VAM reprezentuje číslicový obvod formou grafu, který se skládá z registrů a funkčních uzlů propojených orientovanými hranami. Aplikace je vyvíjena pro využití v rámci projektu HADES [7] (Hazard Detection System) výzkumné skupiny VeriFIT [10] Fakulty informačních technologií VUT Brno.

Důvodem vypracování této bakalářské práce je poskytnutí možnosti formálně verifikovat návrh hardware (převážně procesorů) s využitím vysokoúrovňových návrhových jazyků, jakým je například VHDL. Formální verifikaci je poté možné provést například pomocí nástrojů vyvíjených v rámci projektu HADES. Tyto nástroje pracují právě s grafovou reprezentací zapsanou jazykem VAM, která je výstupem implementované aplikace.

V následujících kapitolách jsou stručně přiblíženy jednotlivé jazyky pro popis hardware, požadavky na funkce a parametry výsledné aplikace a je zde také detailně popsán proces návrhu a implementace aplikace. Z pohledu práce je důraz kladen především na kvalitní interní reprezentaci dat a rozšiřitelnost programu. V poslední části textu jsou shrnuty dosažené výsledky a omezení aplikace (uvedeny různé procesory, které je schopna zpracovat) a diskutovány další možnosti použití, rozšíření a optimalizace výsledné aplikace.

Kapitola 2

Jazyky pro návrh číslicových obvodů

Při procesu vývoje hardware existují různé způsoby reprezentace (popisu) číslicového obvodu, se kterou vývojáři HW pracují. Některé způsoby reprezentace hardware budou v této a v následující kapitole přiblíženy.

V praxi bývá číslicový obvod nejčastěji popsán:

- slovně,
- matematicky,
- graficky pomocí schématu,
- programovacím jazykem.

Rostoucí složitost číslicových zařízení vedla ke vzniku jazyků pro popis hardware (angl. *Hardware Description Languages, HDL*). Ty na rozdíl od schématu umožňují popsat funkci číslicového obvodu jazykem. Zařízení popsané v HDL lze modelovat, simulovat a pomocí procesu syntézy transformovat do prvků cílové architektury, jíž může být např. rekonfigurovatelný obvod typu FPGA. Syntéza je tedy proces analogický kompilaci používané u běžných programovacích jazyků. V současné době v praxi převládají dva jazyky pro popis hardware – VHDL a Verilog.

HDL jazyky zpravidla popisují číslicové obvody na tzv. úrovni přesunů dat mezi registry (angl. *Register Transfer Level, RTL*). Tento přístup předpokládá, že se obvody skládají ze dvou základních typů stavebních bloků – registrů a kombinační logiky. Registry představují základní paměťový prvek a obvykle synchronizují operace obvodu s hranami řídících hodin. Kombinační logika provádí všechny logické funkce obvodu a obvykle se skládá z logických hradel (např. AND nebo OR).

2.1 Jazyk VHDL

Jazyk VHDL (*Very-High-Speed Integrated Circuit HDL)* hierarchicky popisuje číslicová zařízení a jejich části pomocí tzv. *komponent* a *entit*. Ty mohou popisovat celý složitý obvod, popřípadě i jeho část, a výsledný obvod může být být vytvořen vhodnou hierarchickou kombinací komponent a entit. Jako příklad jedné z nejjednodušších částí obvodu můžeme uvést invertor.

Rozhraní komponent a entit se skládá ze signálů a generických parametrů. Signály se mohou podle směru šíření dat vyskytovat v módech IN, OUT nebo INOUT a mohou mít rozdílnou šířku (jeden vodič, nebo sada vodičů). Funkci a chování entity definuje *architektura*, která je vždy svázána s určitou entitou. Architektura může obsahovat vnitřní signály, které slouží pro komunikaci jednotlivých částí této architektury (procesů, dílčích komponent nebo entit). V jazyce VHDL může být architektura popsána třemi různými způsoby, které lze vzájemně kombinovat:

- strukturální popis,
- popis na úrovni datového toku,
- „behaviorální“ popis (popis chování).

Při popisu chování (tzv. *behaviorální popis*) je architektura složena z tzv.*procesů*. Ty sekvenčně popisují způsob, kterým se vstupy dané části obvodu mění na výstupy. Z tohoto popisu tedy nemusí být zřejmá hardwarová reprezentace komponenty (což může někdy způsobovat problémy při syntéze hardware). Procesy ve VHDL nabízejí příkazy pro pozastavení procesu - příkazy *wait, wait_until* apod., a také řídící konstrukce pro podmíněné vykonávání příkazů a cykly známé z běžných programovacích jazyků (*if ... then ..., while ... do ..., for ... loop ...*)

Popis na úrovni datového toku (tzv. *data-flow popis*) modeluje datové závislosti mezi signály pomocí přiřazování hodnot. Příkladem přiřazovacího příkazu může být výraz `Y <= NOT (A AND B)`, který signálu Y přiřazuje negovanou hodnotu výrazu A AND B.

Strukturní popis architektury umožňuje hierarchické znovuvyužití již existujících entit a udává, z jakých částí se daná architektura skládá. Tento způsob popisu může mít více úrovní hierarchie, každá architektura se může skládat z více entit popsaných opět strukturálně, případně behaviorálně. Popis architektur na nejnižší úrovni je vždy behaviorální nebo na úrovni datového toku.

Pro překlad jazyka VHDL do kvalitní a vzhledem k cílům této práce použitelné interní reprezentace v dnešní době neexistuje žádný volně dostupný překladač. Nekomerční projekty [20, 3, 21], které si v průběhu let kladly za cíl vytvoření takového překladače, většinou skončily neúspěšně nebo podporují pouze určitou podmnožinu jazyka VHDL. V současnosti nejpoužívanějším komerčním překladačem je nástroj vyvíjený firmou Verific [23], který je řadou výrobců využíván v rámci jiných vývojových prostředí, např. Xilinx ISE Design Suite.

2.2 Jazyk Verilog

Verilog je druhým z dvojice nejrozšířenějších jazyků pro popis hardware na úrovni RTL. Je rozšířen především v USA, zatímco použití VHDL převládá v Evropě. Jazyk Verilog je používán pro návrh a verifikaci číslicových i analogových obvodů. Možnosti Verilogu jsou obecně srovnatelné s možnostmi jazyka VHDL popsanými výše. Proto v této kapitole uvedeme pouze stručný popis tohoto jazyka a příklad obvodu popsaného jazykem Verilog. V této kapitole se dále zaměříme také na rozdíly mezi jazyky VHDL a Verilog.

Základním datovým typem ve Verilogu je bitový vektor, který může nabývat čtyř různých hodnot. Kromě hodnot logické nuly a logické jedničky je mezi nimi také stav, kdy nás hodnota vektoru „nezajímá“, reprezentovaný znakem X a stav vysoké impedance značený Z. Bitové vektory mohou být ve zapsány v binární, šestnáctkové i desítkové soustavě. Paměťovou jednotkou v jazyce Verilog je datový typ `reg`, který je schopen uchovávat jeden bit.

Chceme-li reprezentovat vícebitový registr, je možné vytvořit pole jednobitových registrů `reg[7:0]`. Pro propojení registrů jsou použity vodiče datového typu `wire`, které můžeme stejně jako registry sdružovat do polí. Pokročilejší paměťovou jednotkou je paměť, kterou můžeme vytvořit jako pole registrů.

Jako příklad jednoduchého obvodu navrženého v jazyce Verilog zde uvedeme zapojení osmibitového registru `R0`. Datové výstupy tohoto registru (vodiče `R0_Q`) jsou invertovány a opět připojeny na datový vstup registr (vodiče `R0_D`). Zapisování hodnot ze vstupních vodičů do registru je synchronizováno hodinovým signálem `CLK`. Pokud je povolen zápis do registru (tj. jeho *write enable* je v logické „1“), dojde při každé náběžné hraně hodinového signálu k přepsání hodnoty zapsané uvnitř registru a její propagaci na výstup tohoto registru.

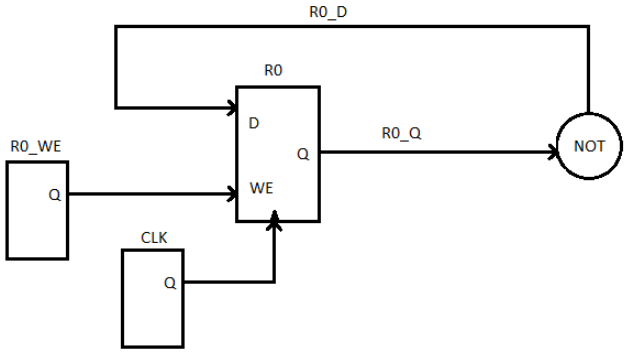
```
module main;

reg   CLK;            // deklarace jednobitových registrů CLK
reg   R0_WE;          // a write enable pro registr R0

reg[7:0]   R0;        // deklarace registru R0
wire[7:0]  R_D;       // a jeho vstupních a výstupních vodičů
wire[7:0]  R0_Q;

always @( posedge CLK) begin  // synchronizace zápisu do registru
                              // s náběžnou hranou CLK

   if((R0_WE == 1'b 1)) begin // pokud je WE pro R0 v log. 1,
     R0 <= R0_D;              // můžeme zapsat hodnotu ze vstupu do registru
   end

end

assign R0_Q = R0;     // propojení registru a jeho výstupních vodičů

assign R0_D =~ R0_Q;  // přivedení invertovaného výstupu na vstup registru

endmodule
```

Příklad 2.1: Jednoduchý obvod popsaný jazykem Verilog.

Z příkladu vidíme, že události synchronizované nástupnou hranou některého ze signálů (v našem příkladu `CLK`) se zapisují do bloku začínajícího `always @()`. Takzvané spojité propojení signálů a registrů píšeme mimo tyto bloky uvozené klíčovým slovem `assign`.

Ačkoli jazyky VHDL a Verilog nabízejí podobné možnosti popisu hardware, poměrně značný rozdíl je v syntaxi obou jazyků. Verilog je bližší syntaxi jazyka C, zatímco syntaxe VHDL se podobá spíše jazyku Ada. Popis prvku v jazyce VHDL je obvykle delší, než jeho ekvivalent v jazyce Verilog.

Jazyk VHDL na rozdíl od Verilogu nabízí množství různých datových typů a také podporuje možnost vytváření uživatelských datových typů. Verilog obsahuje pouze několik základních datových typů – `net` pro reprezentaci vodiče a `reg` pro reprezentaci registru. VHDL



Obrázek 2.1: Schematické znázornění obvodu popsaného jazykem Verilog.

patří mezi silně typované jazyky, zatímco Verilog by se dal spíše popsat jako slabě typovaný (Verilog např. dovoluje přiřazování mezi signály různé bitové šířky).

Výhodou VHDL oproti Verilogu je podpora knihoven, které umožňují uložení již zkompilovaných komponent. VHDL také podporuje vytváření tzv. balíčků (`packages`), které poskytují možnost uložení funkcí a procedur. Knihovny a balíčky následně umožňují znovupoužití svého obsahu v rámci různých modelů.

Hardware navržený v jazyce Verilog je možné převést do jazyka VHDL pomocí nástroje Verilog2VHDL. Možnost zpětného převodu poskytuje další nástroj VHDL2Verilog. Oba konvertory jsou dostupné na webu EDA Utilities [12]. Dalším dostupným konvertorem je aplikace Vhd2vl [9].

Mezi prostředky použitelné pro vývoj hardware patří například ModelSIM [17], který poskytuje simulační prostředí pro HDL jazyky (kromě VHDL a Verilogu také například SystemC a SystemVerilog). Dalšími komerčními nástroji pracujícími s HDL jsou Aldec Active-HDL [1] a Xilinx Vivado [27]. Mezi nekomerční prostředky patří překladač a simulátor IcarusVerilog [24]. Ten převádí kód z jazyka Verilog do interní reprezentace ve formátu `VVP`, kterou poté dokáže simulovat.

2.3 Jazyk CodAL

Jazyk CodAL popisuje číslicové obvody na vyšší úrovni než jazyky VHDL a Verilog a je určen zejména pro popis architektury procesorů. Původně byl vyvíjen v rámci výzkumného projektu Lissom na FIT VUT Brno a jeho cílem je usnadnit plynulý a rychlý vývoj hardware.

Jazyk CodAL patří mezi tzv. smíšené jazyky pro popis architektury — umožňuje popsat jak instrukční sadu, tak mikroarchitekturu navrhovaného procesoru. Výsledný popis procesoru je poté pomocí řetězce podpůrných nástrojů převeden do jiného jazyka pro popis hardware (VHDL nebo Verilog), z něhož je možno navržený hardware syntetizovat. Výhodou jazyka CodAL je také možnost převést návrh do jazyka VAM, který je dále možné analyzovat nástrojem HADES. Jazyk VAM je podrobněji přiblížen v následující kapitole tohoto dokumentu.

2.4 Formát VVP

Formát VVP je reprezentace hardware vytvářená kompilátorem a simulátorem Icarus Verilog [24]. Jazyk VVP popisuje hardware pomocí jednoúrovňové (nehierarchické) sítě základních elementů (angl. *netlist*), z nichž se obvod skládá. Výhodou formátu VVP je jeho snadnější analýza díky tomu, že hierarchický model hardwarového systému z jazyka Verilog je ve VVP zploštěn do podoby netlistu.

Soubor formátu VVP [25] vytvořený překladačem Icarus Verilog se skládá ze čtyř základních částí. První z nich je hlavička, která poskytuje informace o verzi nástroje, s níž byl tento soubor vytvořen, a o modulech potřebných pro simulaci obvodu. Následují dvě části popisující samotný obvod, které budou podrobněji popsány níže. V poslední části se nachází jména zdrojových souborů jazyka Verilog, z nichž byl výsledný VVP soubor vytvořen. První a poslední oddíl zdrojového kódu nejsou pro analýzu hardware prováděnou v rámci této bakalářské práce podstatné, a proto byly z následujícího příkladu zcela vypuštěny.

Jednoduchá ukázka zdrojového kódu formátu VVP byla vytvořena z výše uvedeného příkladu 2.1 obvodu popsaného jazykem Verilog. Obě reprezentace obvodu jsou ekvivalentní.

```
// deklarační část
S_0x880c028 .scope module, "main" "main" 2 17;
 .timescale 0 0;
L_0x882c9c0 .functor NOT 8, v0x882c730_0, C4<00000000>,
                            C4<00000000>, C4<00000000>;
v0x880c770_0 .var "CLK", 0 0;
v0x882c730_0 .var "R0", 7 0;
v0x882c7a8_0 .net "R0_D", 7 0, L_0x882c9c0;  1 drivers
v0x882c828_0 .net "R0_Q", 7 0, v0x882c730_0;  1 drivers
v0x882c8a0_0 .var "R0_WE", 0 0;
E_0x880be88 .event posedge, v0x880c770_0;
```

Příklad 2.2: Deklarační část VVP souboru.

V deklarační části souboru se pro každou instanci modulu jazyka Verilog, z nichž je výsledný obvod složen, nachází jeden oddíl zvaný `scope`. Uvnitř tohoto oddílu jsou deklarovány všechny hardwarové elementy obvodu a také tzv. *události* (angl. *events*), které mohou nastat na některém z vodičů nebo registrů. Tyto události odpovídají podmínkám v blocích `always @()` v jazyce Verilog. Pro ilustraci, událost zachycená v našem příkladu nastává vždy na náběžné hraně registru `CLK`. Zachycení pouze náběžné hrany je deklarováno klíčovým slovem `posedge`. V této části souboru jsou vytvořena spojení mezi prvky, která v původním zdrojovém kódu byla popsána klíčovým slovem `assign` a nebyla synchronizována hodinovým signálem.

Deklarace každého hardwarového prvku zpravidla obsahuje tyto parametry:

- jednoznačný identifikátor v rámci celého souboru (tzv. `label` – návěští), např. `v0x880c770_0`,
- typ elementu, např. `.var` nebo `.net`,
- jméno elementu, tj. původní jméno prvku v jazyce Verilog, pokud bylo pro daný prvek definováno,

- bitovou šířku – uvedenou buď jako jedno kladné číslo, nebo jako rozsah vodičů, např. "7 0" značí datovou šířku osmi vodičů s nejvíce významným sedmým bitem a nejméně významným bitem číslo nula,
- seznam vstupů prvku, který obsahuje unikátní identifikátory dalších elementů (může být i prázdný).

Nejčastějšími prvky, vyskytujícími se v souboru formátu VVP jsou:

- `.var` – vytvořen z registrů `reg` jazyka Verilog,
- `.net` – odpovídající vodičům `wire` ve Verilogu,
- `.functor` – jednotka reprezentující funkční uzel s čtyřmi vstupy, který může vykonávat nějakou logickou operaci, propagovat dále svůj vstup nebo vytvářet konstantní hodnotu, speciální typy functorů mohou sloužit také jako multiplexor.

Mezi další hardwarové elementy generované v rámci VVP patří například `.array` reprezentující paměťovou jednotku, `.cmp` provádějící porovnání na rovnost nebo nerovnost dvou vstupů a `.arith/sum` či `.arith/mul` představující sčítačku a násobičku. Kompletní seznam a popis všech komponent VVP modelu je dostupný v online repozitáři projektu Icarus Verilog [25].

Druhou podstatnou částí VVP souboru z hlediska analýzy hardware je jeho instrukční část. V ní jsou popsána spojení mezi registry a vodiči, která byla ve Verilogu zapsána uvnitř `always @()` bloků. V rámci projektu Icarus Verilog pomocí interpretace této části kódu probíhá simulace daného obvodu.

```
// část pro simulaci - provádění instrukcí
    .scope S_0x880c028;
T_0 ;
    %wait E_0x880be88;
    %load/vec4 v0x882c8a0_0;
    %cmpi/e 1, 0, 1;
    %jmp/0xz  T_0.0, 4;
    %load/vec4 v0x882c7a8_0;
    %assign/vec4 v0x882c730_0, 0;
T_0.0 ;
    %jmp T_0;
    .thread T_0;
```

Příklad 2.3: Simulační část VVP souboru.

Instrukční sada [26] je navržena pro jednoduchou zásobníkovou architekturu simulátoru. Ten obsahuje příznakový registr, několik index registrů a zásobník bitových vektorů. Instrukční část souboru je složena ze samotných instrukcí a deklarací vláken, z nichž každé provádí určitou část těchto instrukcí. Každému bloku `always @()` odpovídá právě jedno vlákno ve VVP běžící v nekonečném cyklu.

Na začátku provádění vlákna se zpravidla nachází instrukce `%wait` s operandem typu `.event`, na níž je vlákno pozastaveno do chvíle, než daná událost nastane. Poté je instrukcí `%load` načtena na zásobník vektorů hodnota určitého VVP prvku a pomocí instrukce `%cmp` jsou nastaveny bity příznakového registru. Tato konstrukce reprezentuje podmínku

`if(R0_WE == 1)` z příkladu jazyka Verilog uvedeného výše. Pokud je podmínka splněna, podmíněný skok instrukce `%jmp0xz` se neprovede, je načtena další hodnota na zásobník a pomocí instrukce `%assign` je tato hodnota uložena do registru.

V případě, že s některou z hodnot je třeba ještě provést určitou operaci, jsou ve VVP k dispozici instrukce pro aritmetické a logické operace jako například `%inv` a `%add` nebo pro výběr pouze určité části bitového vektoru pomocí instrukce `%part`.

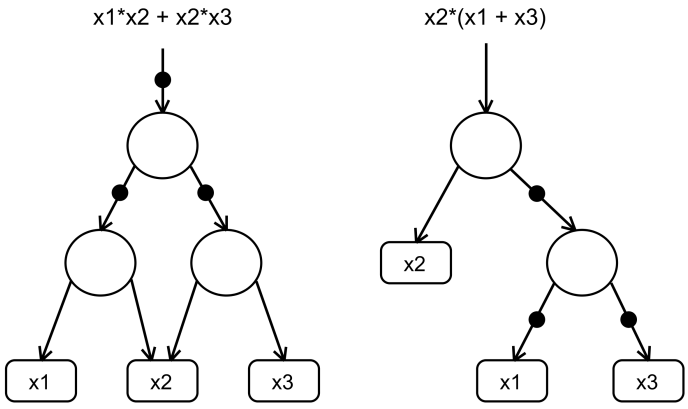
Kapitola 3

Reprezentace hardware pro účely formální verifikace

Protože většina formálních nástrojů (jako například HADES [7], ABC [6] nebo Hector [15]) pro automatickou verifikaci nepracuje přímo s jazyky pro popis hardware jako VHDL nebo Verilog, užívají se takzvané grafy toku dat (angl. *Data-flow Graphs, DFG*). Ty mohou posloužit jako vstup pro verifikační software. Problémem je ovšem převod z návrhu v jazycích HDL do DFG.

3.1 Jazyk AIG (And-Inverter Graph)

AIG neboli *And-Inverter Graph* je orientovaný acyklický graf, který reprezentuje strukturu implementace a funkcionalitu číslicového obvodu. Tento graf se skládá z uzlů představujících logickou konjunkci (tedy operaci AND), tyto uzly mají vždy dva vstupy, pro něž operaci AND provádějí. Listovými uzly v grafu jsou pojmenované proměnné, které slouží jako vstupy pro AND uzly. Uzly jsou spojeny orientovanými hranami, na kterých může být umístěn symbol značící negaci daného vstupu (reprezentuje invertor, tj. operaci NOT). Na obrázku je uveden příklad jednoduchého AIG grafu, ten může pro stejný výraz nabývat různých podob (například upravením výrazu lze celý graf zjednodušit).



Obrázek 3.1: Příklad dvou různých AIG grafů zobrazujících stejný výraz.

Tato reprezentace je vhodná i pro velké obvody díky svému kompaktnímu formátu a navíc se dá uplatnit i ve formální verifikaci. Nevýhodou je ztráta vysokoúrovňové informace, při zpětném převodu už tedy nemusí být zřejmé, které části dříve reprezentovaly registr, apod. Jednou ze sad nástrojů použitelných pro práci s AIG grafy je AIGER [14, 4, 5], který zároveň definuje formát uložení grafové struktury. AIGER je možné použít pro ověřování určitých vlastností modelů. AIG grafy jsou také využívány v soutěži verifikace hardware – Hardware Verification Competition.

3.2 Jazyk SMV (Symbolic Model Verification)

Jazyk SMV je vstupním jazykem nástroje pro verifikaci systémů NuSMV [8]. Tento jazyk byl navržen pro popis systémů s konečným počtem stavů reprezentovaný grafem. Podobně jako u jazyků pro popis hardware může být systém dekomponován do jednotlivých modulů, z nichž každý může být vícekrát instanciován. To umožňuje popsat systém hierarchicky pomocí znovupoužitelných komponent. Moduly mohou být skládány dohromady synchronně i asynchronně. Při synchronní kompozici odpovídá jeden krok systému (jeden pulz synchronizačního signálu) jednomu kroku každé komponenty. U asynchronního spojení modulů může být každý z modulů synchronizován jiným signálem.

Jazyk SMV poskytuje pouze základní datové typy a to:

- boolovský typ nabývající hodnot true nebo false,
- celočíselný datový typ integer
- výčtový typ enum

```
MODULE main
   VAR
     bit0 : counter_cell(1);
     bit1 : counter_cell(bit0.carry_out);
     bit2 : counter_cell(bit1.carry_out);
   SPEC
      AG AF bit2.carry_out;

MODULE counter_cell(carry_in)
   VAR
     value :  boolean;
   ASSIGN
      init(value) := 0;
      next(value) := value + carry_in mod 2;
   DEFINE
     carry_out := value & carry_in;
```

Příklad 3.1: Tříbitový čítač navržený v jazyce SMV.

V příkladu vidíme definici modulu `counter_cell`, který obsahuje jednu bitovou proměnnou `value`. Vstupním parametrem tohoto modulu je bitová hodnota `carry_in`, která je v každém taktu hodinového signálu přičítána k hodnotě proměnné value. Výstupní hodnotou modulu `counter_cell` je hodnota `carry_out`, která je generována vždy, když jsou

hodnoty `carry_in` a `value` obě rovny logické „1“ (tedy vždy, když dojde k přetečení hodnoty `value`).

V modulu `main` je modul `counter_cell` třikrát instanciován. Jednotlivé instance buněk jsou mezi sebou propojeny pomocí vstupních parametrů. V příkladu vidíme, že hodnota nejnižšího bitu čítače se zvyšuje s každým taktem, protože jako vstupní parametr dostává hodnotu 1. Hodnoty čítačů vyšších bitů se inkrementují pomocí hodnot `carry_out` předcházející buňky čítače.

Při verifikaci systému nástrojem NuSMV je postupně prohledáván stavový prostor navrženého systému. Prohledáním celého stavového prostoru daného systému je možné dokázat, zda jsou požadované specifikace systému splněny. Ve své podstatě je jazyk NuSMV grafovou reprezentací zkoumaného systému podobně, jako jazyk VAM popsaný níže.

3.3 Jazyk VAM (Variable Assignment Model)

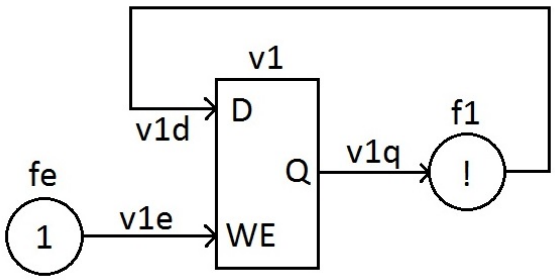
V jazyce VAM (zkratka pro *Variable Assignment Language*) je model RTL návrhu procesoru reprezentován strukturou, která popisuje tok dat v procesoru. Struktura je tvořena sítí funkčních a ukládacích uzlů. Tyto uzly jsou propojeny orientovanými hranami, reprezentujícími tok dat mezi uzly. Celý VAM graf je tzv. typu *next-event*, což znamená, že stav jednotlivých signálů se mění vždy s hranou hodinového signálu.

Ukládací uzly (angl. *storage nodes*) reprezentují hardwarový element, který je schopný zapamatovat si data po určitou dobu. Nejjednodušším příkladem ukládacího uzlu je jednobitový registr, složitějšími uzly mohou být pole registrů, nebo jednotka paměti. Registr je ovládán pomocí povolovacího signálu WE (angl. *write enable*), a má vstupní a výstupní datový signál. Pole registrů a paměti jsou složitějšími strukturami, které obsahují více signálů, například adresu prvku, jejž chceme číst, apod.

Funkční uzly (angl. *functional nodes*) reprezentují logické či aritmetické funkce, které transformují vstupní signály na výstupní. Výstupní signál je v každém čase aktuální pro dané vstupy (přepočítává se ihned). VAM podporuje téměř všechny výrazy a funkce, které jsou dostupné v klasických HDL. Pro zjednodušení při analýze jsou zavedeny tzv. *abstraktní operátory*, kterými se ale v tomto dokumentu nebudeme dále zabývat.

Hrany v grafu reprezentují signály, které mohou přenášet informace z jednoho uzlu do druhého. Parametry hrany jsou název signálu, bitová šířka signálu a typ signálu (např. adresa nebo kontrolní signál).

Popis grafu pomocí jazyka VAM je možné demonstrovat na jednoduchém příkladu, který



Obrázek 3.2: Grafické znázornění příkladu zapsaného pomocí VAM.

popisuje jednobitový registr s datovými porty připojenými přes invertor:

```
// Deklarace signálů v grafu (typ, název, bitová šířka)
(sig v1q 1)
(sig v1d 1)
(sig v1e 1)

// Definice funkčního uzlu fe (typ, název, vstupy, výstupy, popis operace).
// Tento uzel přiřazuje signálu v1e trvalou logickou 1.
(fnode fe (input ) (output v1e) (assign (:= v1e 1)) )

// Definice invertoru f1
(fnode f1 (input v1q) (output v1d) (assign (:= v1d (! v1q))) )

// Definice registru
// v1 (typ, název, bitová šířka, povolovací signál, vstup, výstup)
(reg v1 1 (we v1e) (d v1d) (q v1q) )
```

Příklad 3.2: Jednoduchý obvod navržený v jazyce VAM.

Obvod popsaný tímto zdrojovým kódem je graficky znázorněn na obr. 3.2. V příkladu vidíme deklaraci základního paměťového prvku – registru `v1`. Jeho vstupní signál `v1d` se zapíše na jeho výstup `v1q` pouze v případě, že je psaní do registru povoleno povolovacím signálem registru `v1e`. Hodnota signálu reprezentujícího datový výstup registru je invertována ve funkčním uzlu `f1`, jehož výstupem je vstupní datová hodnota registru.

Kapitola 4

Specifikace požadavků na překladač

Zadání bakalářské práce a obecné zásady kvalitního návrhu a implementace software kladou na výslednou aplikaci určité nároky, které budou shrnuty v této kapitole.

Volně dostupně zatím neexistuje nástroj, který by formálně verifikoval hardware popsaný některým z jazyků Verilog / VHDL. Proto se pro verifikační účely používá grafové reprezentace. Procesory navržené v jazyce CodAL lze existujícími programy převést do jazyka VAM, což umožňuje následné ověření modelu. Pro VHDL ani Verilog software převádějící hardware do VAM reprezentace zatím není k dispozici. Zpřístupnění takového nástroje si za úkol klade právě tato práce.

Cílem této bakalářské práce je vyvinout software, který bude překládat popis hardware do vnitřní reprezentace, tudíž aplikace musí být schopna zpracovat návrh nejrůznějších digitálních obvodů v jazyce VHDL a tento návrh převést na abstraktní syntaktický strom, ze kterého je možné posléze generovat požadovanou výstupní reprezentaci. Pro tyto účely není nutné vyvíjet celý překladač jazyků VHDL / Verilog, ale lze použít některý z již vytvořených syntetizátorů. V rámci této práce je použit syntetizátor IcarusVerilog [24], který převádí návrh HW v jazyce Verilog do formátu `VVP`. Tento formát je pro navrhovaný překladač výhodný, protože zachovává vysokoúrovňovou informaci a zároveň zplošťuje hierarchický popis hardware. Protože nástroj Icarus Verilog zpracovává návrhy hardware popsané jazykem Verilog, je nutné popisy procesorů z jazyka VHDL nejprve do Verilogu převést. K tomu je v rámci vývoje tohoto překladače použit nástroj vhd2vl [9], jehož výstup může být dále zpracováván simulátorem Icarus Verilog.

Důležitým bodem zadání je vytvoření kvalitní interní reprezentace zpracováním abstraktního syntaktického stromu, která bude uchovávat veškeré důležité informace o obvodech nutné pro další zpracování (např. formální verifikaci).

Vzhledem k tomu, že překladač je koncipován jako podpůrný nástroj pro verifikační prostředí HADES, je požadována výstupní reprezentace v jazyce VAM. Překladač by měl ovšem být napsán dostatečně obecně, aby umožňoval také možnost poskytnout základ programu pro převod do jiné koncové reprezentace (např. jiného druhu DFG). To by měl umožnit kvalitní interní model.

Součástí vypracování této technické zprávy je také sada testů pro výslednou aplikaci, která ověří správnost jednotlivých komponent překladače i použitelnost překladače jako celku.

Kapitola 5

Návrh implementace překladače

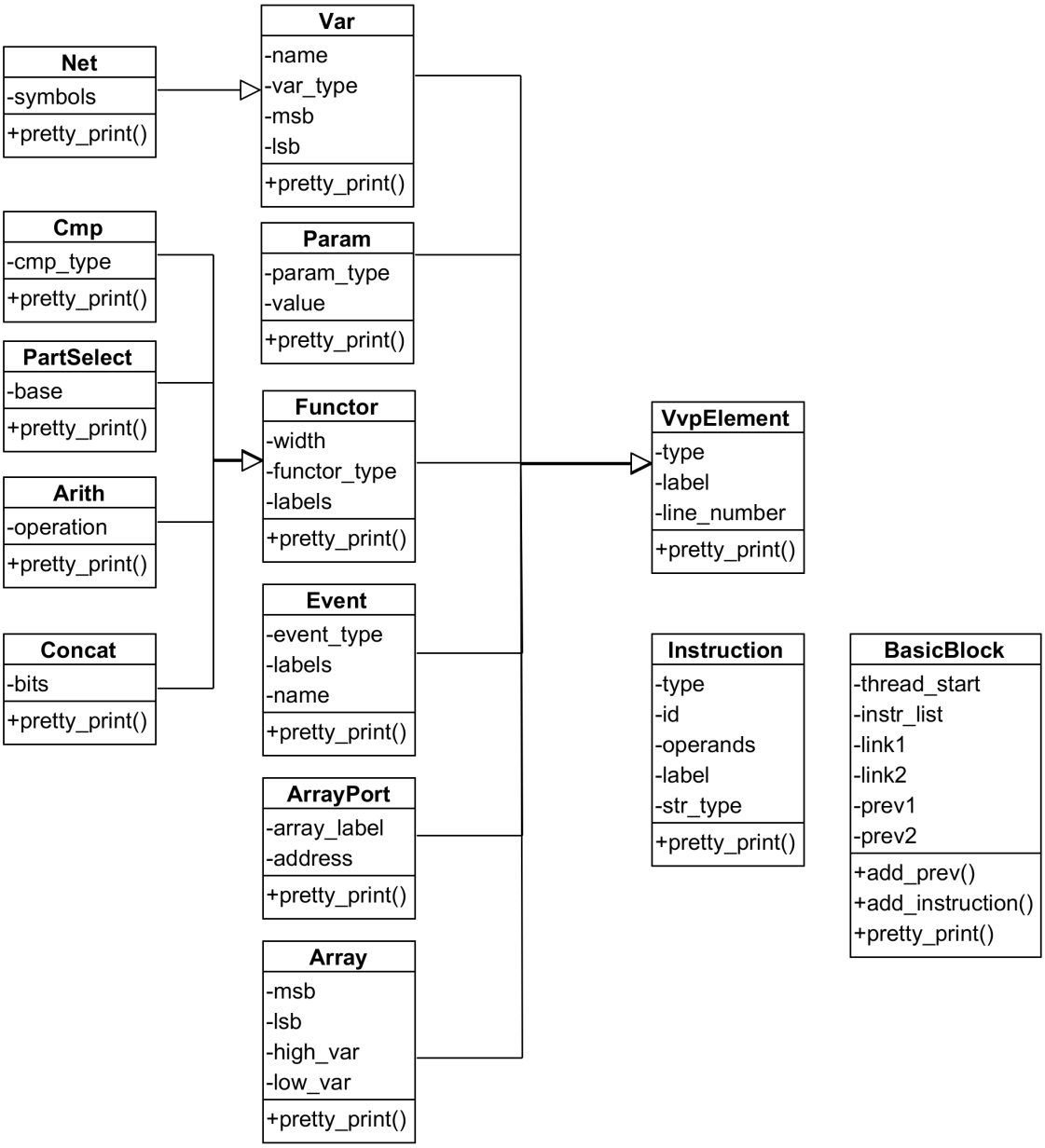
5.1 Zpracování vstupních dat

Navržený překladač zpracovává soubory ve formátu `VVP`, které jsou výstupem nástroje IcarusVerilog. Tyto soubory analyzuje pomocí lexikálního analyzátoru. Ten převede vstupní soubor na řetězec terminálů (tzv. tokenů). Výstup lexikálního analyzátoru je syntaktickým analyzátorem sestaven do abstraktního syntaktického stromu. Diagram tříd použitý pro reprezentaci tohoto abstraktního syntaktického stromu je zobrazen na obrázku 5.1. Při rozpoznání pravidla deklarujícího určitý prvek VVP modelu je syntaktickým analyzátorem zavolána příslušná metoda překladače, která daný prvek vytvoří. Ke všem prvkům je možné přistupovat jednotně přes rozhraní definované třídou `VvpElement`. Každá instance je jednoznačně identifikovaná svým návěštím ze zdrojového VVP souboru.

Třídy reprezentující VVP prvky obsahují atributy uchovávající unikátní identifikátory objektů, s nimiž je daná instance třídy propojena. Tyto atributy jsou typu řetězec, protože v rámci VVP souboru může být určitá komponenta odkazována svým identifikátorem ještě předtím, než byla sama deklarována. Samotné spojení všech elementů modelu pomocí ukazatelů je proto provedeno až po zpracování celého vstupního souboru a převedení zkoumaného obvodu do jiného objektového modelu, který je popsán v následující sekci této kapitoly.

Jednotlivé třídy interního modelu reprezentují struktury formátu VVP a jejich atributy uchovávají informace o těchto strukturách a vztazích mezi nimi. Následuje stručný popis jednotlivých tříd:

- `VvpElement` – třída nadřazená všem objektům VVP modelu, obsahuje proměnnou obsahující typ daného objektu a unikátní identifikátor objektu, deklaruje virtuální metodu `pretty_print()`, již musí všechny všechny třídy odvozené z třídy VvpElement implementovat.
- `Var` – bitový vektor, který propaguje svůj vstup do objektu třídy `Functor`, do `Var` může být zapisována hodnota pomocí instrukcí behaviorálního popisu.
- `Net` – bitový vektor podobný `Variable`, nemůže do něj však být zapisována hodnota pomocí instrukcí.
- `Param` – reprezentuje pojmenovanou konstantu, tzv. parametr ve Verilogu.
- `Functor` – základní logická jednotka, kombinuje až 4 vstupy do jednoho výstupu a provádí nad nimi nejčastěji logickou operaci (například logický součin).



Obrázek 5.1: Třídní diagram VVP modelu navrhovaného překladače.

- `Event` – slouží pro synchronizaci vlákna podle určité události, například vzestupné hrany signálu.
- `Arith` – objekt reprezentující uzel provádějící aritmetickou operaci se dvěma VVP prvky (například sčítání nebo násobení).
- `Concat` – prvek sloužící ke konkatenaci (sjednocení) několika vodičů.
- `PartSelect` – objekt sloužící pro výběr pouze některých bitů z původní šířky vodiče (oříznutí vodiče).
- `Cmp` – slouží pro test na shodu hodnot dvou objektů.
- `Array` – reprezentuje paměťovou jednotku (pole registrů), má definované čtecí porty a adresové vodiče.
- `ArrayPort` – čtecí port, který je vždy přiřazen k určité paměti a je adresován některým vodičem.
- `VvpInstruction` – instrukce, které jsou součástí behaviorálního popisu obvodu.
- `BasicBlock` – skupina instrukcí, která se provádí jako celek, podrobněji je popsána v pozdější části této kapitoly.
- `Init` – třída popisující inicializaci určitého VVP prvku na konkrétní hodnotu před začátkem simulace daného obvodu.

Třídní diagram vizualizující hierarchii tříd VVP modelu a jejich atributy je znázorněn na obrázku 5.1.

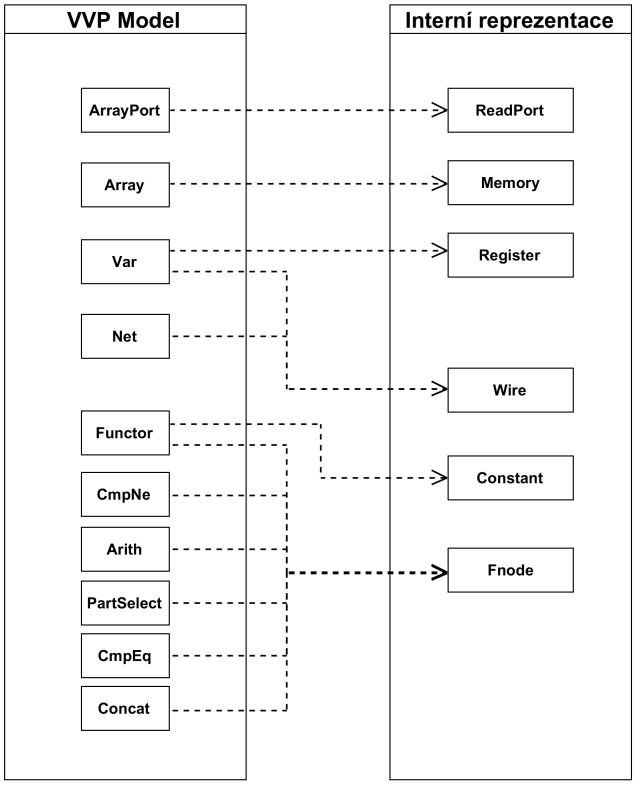
5.2 Vytvoření interního objektového modelu

Z důvodu poměrně komplexního modelování systému formátem VVP byl navržen druhý objektový model pro interní reprezentaci hardwarového obvodu v rámci vytvářeného překladače. Tento model zahrnuje i interpretované VVP instrukce realizující behaviorální popis obvodu. Nový interní model by také měl být bližší cílové grafové reprezentaci v jazyce VAM.

Po zpracování celého vstupního souboru je proto pro každý původní prvek modelu VVP vytvořena nová instance patřící do interního objektového modelu. V rámci tohoto modelu už jsou veškeré objekty propojeny nikoli pomocí identifikátorů, nýbrž pomocí ukazatelů. Dalším změnou oproti původní VVP reprezentaci je fakt, že objekty obsahují také seznamy svých výstupních objektů, což usnadňuje další analýzu a optimalizace modelu.

Většina VVP elementů je převedena na své interní ekvivalenty tak, že jednomu VVP objektu odpovídá jeden objekt interního modelu. Výjimku tvoří instance třídy `Var`, které jsou převedeny na ekvivalentní registr, k němuž je navíc ihned vytvořen vodič reprezentující výstup registru. K tomuto vodiči jsou poté připojovány všechny objekty, pro něž je registr vstupem. Díky tomuto kroku jsou registry s ostatními interními objekty propojeny pomocí vodičů.

Druhou výjimkou jsou instance třídy `Functor`. Tyto funkční uzly mají ve VVP reprezentaci vždy čtyři vstupy. Pokud některý ze vstupů není využit, je na něj přivedena konstanta, která neovlivňuje výslednou hodnotu generovanou uzlem. Pro příklad nulová



Obrázek 5.2: Diagram znázorňující transformaci VVP modelu na interní model.

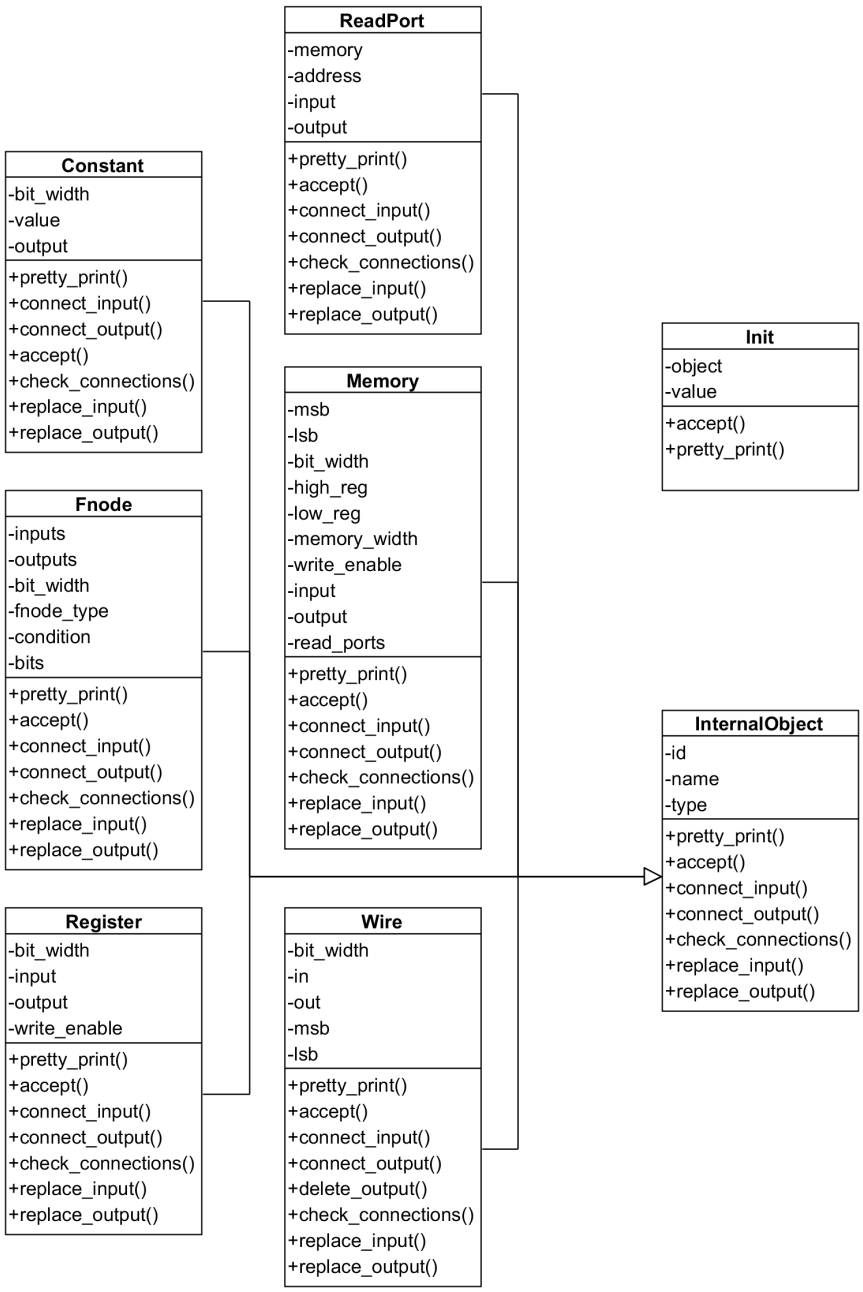
konstanta s šířkou pěti bitů je ve VVP zapsána jako `C4<00000>`. Pro zjednodušení modelu jsou při převodu funktorů na jejich interní reprezentaci tyto konstanty analyzovány. Pokud je možné konstantu vypustit bez změny výsledné hodnoty generované funktorem, je tak učiněno. V opačném případě je zároveň s funkčním uzlem vygenerován také objekt reprezentující danou konstantu a ten je jako vstup připojen k uzlu.

5.3 Analýza a interpretace instrukcí

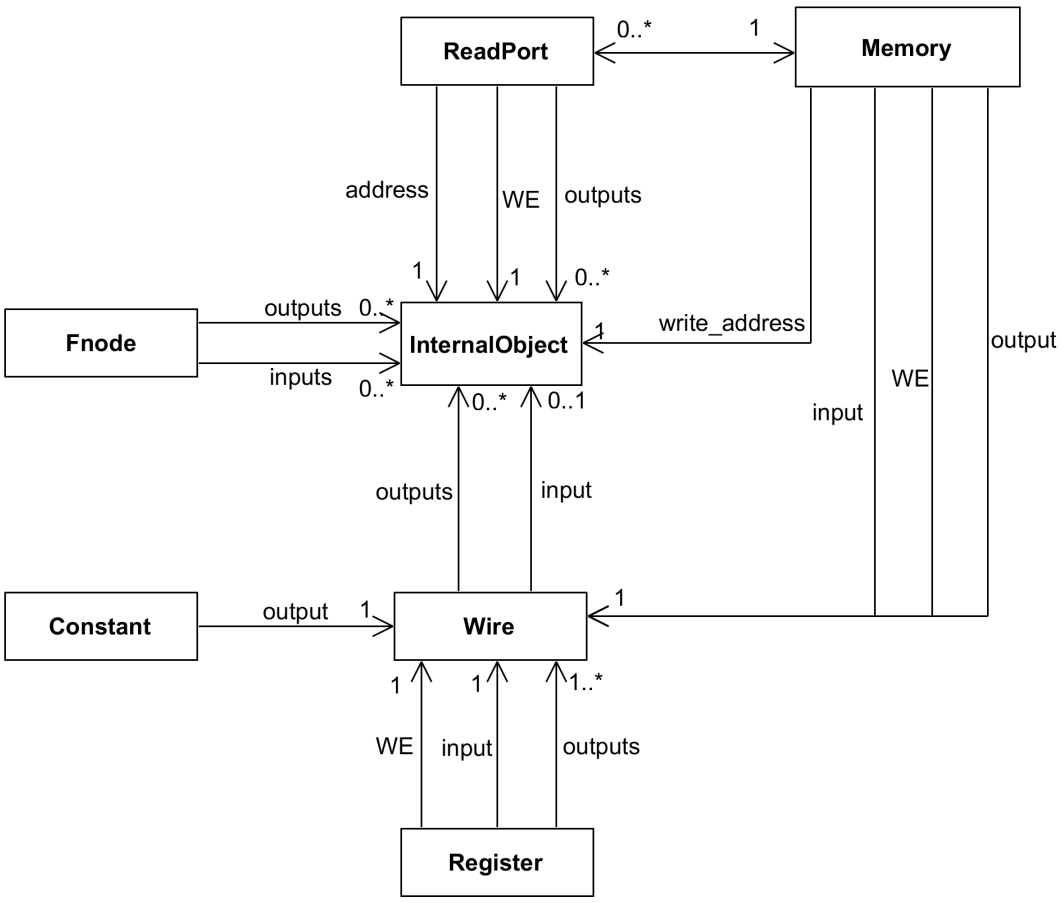
Poté, co je na interní model převedena deklarační část VVP souboru, je nutné do modelu zahrnout také instrukční oddíl VVP formátu popisující chování obvodu imperativně. Tato část zdrojového kódu se většinou týká zapisování hodnot do registrů či pamětí, které je povolováno vodičem `write enable` a synchronizováno hodinovým signálem. Pro tuto činnost je třeba pomocí objektů interního modelu vytvořit reprezentaci, která bude plnit ekvivalentní činnost.

Z důvodu interpretace jsou instrukce rozděleny do tzv. základních bloků. Základní blok [22] (angl. *basic block*) je posloupnost maximálního počtu instrukcí taková, že platí:

- vstupní bod řízení je na prvním příkazu,
- výstupní bod řízení je na posledním příkazu,
- příkazy se provádí vždy sekvenčně v pořadí daném posloupností.



Obrázek 5.3: Třídní diagram interního modelu navrhovaného překladače.



Obrázek 5.4: Diagram závislostí tříd interního modelu.

Základní blok může obsahovat skokovou instrukci pouze na konci. Blok instrukcí ve VVP formátu může končit buďto návěštím, nebo právě skokovou instrukcí. V případě, že se jedná o podmíněný skok, může se pokračovat na dva různé bloky, na blok který je cílem skoku nebo na následující základní blok. V ostatních případech se může přejít pouze do jednoho bloku (cíle nepodmíněného skoku nebo následujícího bloku v případě návěští.)

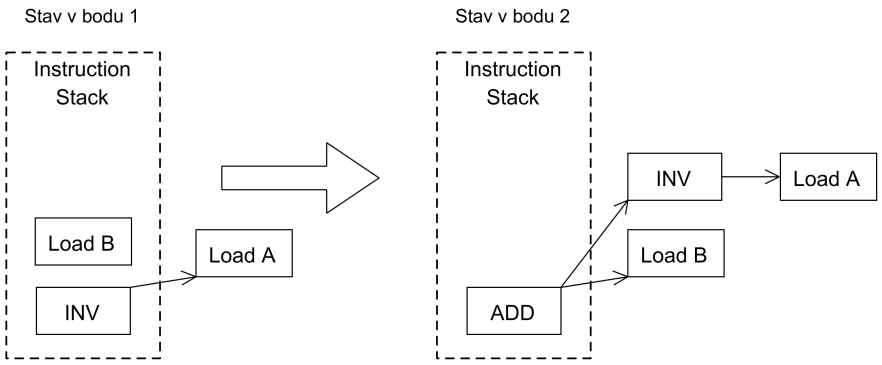
Po analýze instrukcí je pro každý nalezený základní blok vytvořena instance třídy `BasicBlock`, která obsahuje seznam referencí na všechny instrukce tohoto bloku, a také informace o tom, na které bloky se ze stávajícího bloku může přejít. Navíc jsou zde zahrnuty i ukazatele na bloky, ze kterých se do daného bloku může vstoupit. Tyto ukazatele napomáhají dalším analýzám popsaným níže.

5.3.1 Interpretace hodnot datových vstupů registrů a pamětí

Zápis do registru nebo do paměti se provádí pomocí instrukcí `%store` a `%assign`. Tyto instrukce do registru nebo buňky paměti zadané operandem vloží vrchol zásobníku. Interpretace instrukcí tedy spočívá ve zjištění hodnoty, která se nachází na vrcholu zásobníku bitových vektorů ve chvíli provedení těchto příkazů. Dále je v této sekci popsán princip interpretace instrukcí v navrhovaném překladači.

Pro každou instrukci je nalezen základní blok, ve kterém se tato instrukce nachází. Jeden základní blok může obsahovat i více instrukcí `%store` nebo `%assign`, které jsou zpracovávány zvlášť. Aby proto nedocházelo k vícenásobnému generování stejných objektů je nutné interpretací zjistit, které příkazy tohoto bloku se podílejí na výsledné hodnotě zásobníku, která se bude přiřazovat do daného registru. Pouze pro tyto instrukce se poté generuje interní reprezentace. Pro interpretaci jsou VVP instrukce rozděleny do pěti základních skupin:

- instrukce, které nepracují se zásobníkem – v této chvíli se ignorují (např. `%wait`),
- příkazy, které na zásobník ukládají bitový vektor (např. `%load`),
- instrukce, které ze zásobníku vybírají bitový vektor (např. `%store`),
- instrukce, které ze zásobníku vyberou jeden operand, provedou nad ním určitou operaci a výsledek vrátí na zásobník (např. `%inv` – logické invertování hodnoty),



Obrázek 5.5: Stavy zásobníku bitových vektorů při interpretaci instrukcí.

- instrukce, které z vrcholu zásobníku vyberou dva operandy, provedou nad nimi operaci a výsledek uloží zpět na zásobník (např. `%add` – součet dvou hodnot z vrcholu zásobníku).

Během interpretace je použit zásobník instrukcí, na nějž se podle výše uvedených skupin aplikují jednotlivé instrukce. Příkazy, které provádějí operace nad hodnotami na zásobníku a výsledek vracejí zpět na zásobník, si ukládají seznam instrukcí, jež vedly ke právě zpracovávané hodnotě. Ve chvíli, kdy interpretace narazí na aktuálně zkoumanou instrukci `%store` nebo `%assign`, je na vrcholu zásobníku instrukce vytvářející výslednou hodnotu spolu se seznamem všech instrukcí, které se na této hodnotě podílely. Operace se zásobníkem a vztahy mezi instrukcemi jsou uvedeny v příkladu č. 5.1 a znázorněny graficky na obrázku č. 5.5.

```
T_0:
    %load A;
    %inv;
    %load B;       // bod 1 - po interpretaci instrukce %load
    %add;          // bod 2 - po interpretaci instrukce %add
    %assign X;
```

Příklad 5.1: Základní blok s interpretovanými instrukcemi.

Podle výsledného seznamu instrukcí jsou poté vygenerovány instance interního objektového modelu. Po správném propojení těchto objektů je výstup objektu reprezentujícího poslední instrukci přiveden na datový vstup registru.

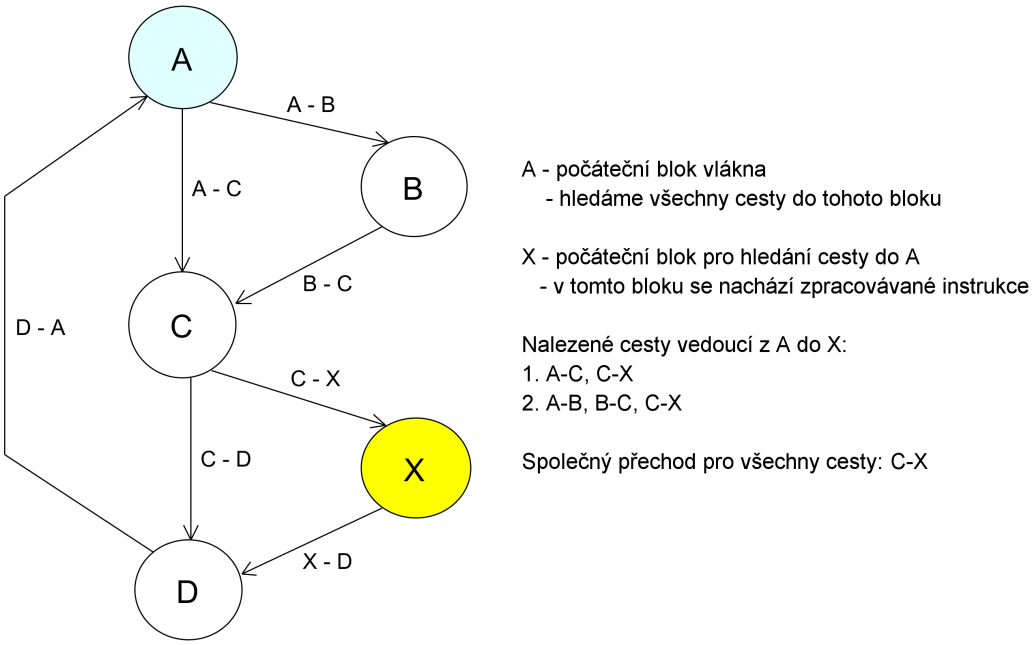
5.3.2 Analýza povolovacích vstupů registrů a pamětí

Kromě již popsané interpretace datového vstupu registru je také nutné zjistit kombinaci podmínek, kterou je povolen zápis do registru (`write enable`). Proto je třeba analyzovat přechody mezi jednotlivými základními bloky.

Pro blok, v němž se nachází analyzovaná instrukce, nalezneme pomocí zpětného prohledávání algoritmem DFS (*Depth First Search* [2]) všechny seznamy základních bloků, které může vlákno na cestě od svého začátku k danému bloku projít. Algoritmus DFS pracuje následovně:

1. sestrojte zásobník, který bude obsahovat všechny uzly určené k expanzi a do něj umístěte počáteční uzel,
2. je-li zásobník prázdný, pak ukončete prohledávání, úloha nemá řešení,
3. vyberte uzel z vrcholu zásobníku,
4. je-li uzel cílovým uzlem, ukončete prohledávání a jako řešení vraťte cestu mezi kořenovým a cílovým uzlem,
5. vybraný uzel expandujte a všechny jeho bezprostřední následníky umístěte na zásobník, pokračujte bodem 2.

V rámci této bakalářské práce je výše popsaný algoritmus použit s drobnými úpravami. Prohledávání nekončí při první nalezené cestě, ale cesta se pouze přidá do seznamu výsledků a algoritmus pokračuje v činnosti, dokud test na prázdnost zásobníku v bodě 2 cyklus



Obrázek 5.6: Příklad hledání cest pro základní bloky.

neukončí. Po skončení je vrácen seznam všech nalezených cest. Tento seznam je poté ze své současné reprezentace v podobě posloupnosti základních bloků převeden na posloupnost přechodů mezi bloky ve formě počáteční blok - koncový blok. Ze všech seznamů možných přechodů je nakonec vytvořen průnik, tzn. zůstanou pouze přechody mezi bloky, které jsou pro všechny cesty společné. A právě provedení těchto přechodů je postačující k tomu, aby zkoumaná instrukce zapsala příslušnou hodnotu do registru. Prohledávání cest v grafu základních bloků předpokládá vlákno vykonávající instrukce v nekonečné smyčce. Případ, kdy je vlákno po vykonání požadovaných instrukcí ukončeno, je popsán níže v této sekci v části 5.3.3. Proces vyhledávání cest v grafu je ilustrován na obrázku 5.6.

Výsledkem prohledávání znázorněného na obrázku je seznam přechodů mezi bloky, které se musí provést, aby se vlákno dostalo na místo analyzované instrukce a tuto instrukci provedlo. Analýzou nalezených přechodů tedy můžeme zjistit, jaké objekty interního modelu ovlivňují zápis do daného registru a ve chvíli, kdy tyto objekty nabývají požadovaných hodnot, přivést na povolovací vstup registru logickou „1“. Tím je zápis do registru povolen a vloží se do něj hodnota přivedená po interpretaci instrukcí na jeho datový vstup. Analyzované přechody mezi bloky mohou proběhnout třemi různými způsoby:

- podmíněným skokem,
- nepodmíněným skokem,
- sekvenčním přechodem do dalšího bloku.

Poslední dva přechody nezávisí na hodnotách objektů, z nichž se návrh obvodu skládá a provedou se vždy. Proto se v analýze zaměřujeme pouze na přechody provedené podmíněným skokem. Provedení nebo neprovedení podmíněného skoku vždy závisí na stavu příznakového registru (angl. *flag register*). Hodnota příznakového registru je ovlivňována

instrukcemi `%cmp` a `%flag_set`. Tyto instrukce vybírají vrchol zásobníku bitových vektorů a podle jeho hodnoty nastaví příznakový registr.

```
   ...
T_0:     // základní blok 1
    %load/vec4 R0_WE;      // načtení hodnoty vodiče R0_WE na zásobník
    %cmpi/e 1, 0, 1;       // porovnání vrcholu zásobníku s hodnotou 1
    %jmp/0xz  T_0.0, 4;    // podmíněný skok na návěští T_0.0
         // základní blok 2
    %load/vec4 R0_D;       // načtení hodnoty R0_D na zásobník
    %assign/vec4 R0;       // přiřazení vrcholu zásobníku do registru R0
T_0.0:   // základní blok 3
   ...
```

Příklad 5.2: Podmíněný skok v instrukční části VVP souboru.

V příkladu vidíme, že uskutečnění podmíněného skoku na návěští `T_0.0` závisí na hodnotě vodiče `R0_WE` a výsledku jejího porovnání s hodnotou logické „1“. Pokud se na vodiči `R0_WE` nachází hodnota logické „1“, podmíněný skok na návěští `T_0.0` se neprovede a vlákno pokračuje do bloku 2, ve kterém načte hodnotu vodiče `R0_D` a přiřadí ji do registru `R0`. Abychom v interním objektovém modelu zajistili stejné chování zpracovávaného obvodu, vytvoříme podmínkový funkční uzel, který na povolovací vstup registru `R0` přivede logickou „1“ vždy, když se bude vodič `R0_WE` nacházet v logické „1“.

Tuto analýzu přechodů mezi bloky musíme dále zobecnit, aby bylo možné zpracovávat i složitější podmínky pro zápis hodnoty do registru. Jak už bylo zmíněno výše, instrukce `%cmp` při nastavení příznakového registru pracuje s vrcholem zásobníku bitových vektorů. Abychom zjistili, jaká hodnota se na vrcholu zásobníku právě nachází, zopakujeme pro instrukci `%cmp` stejný proces interpretace instrukcí, který již byl popsán výše pro instrukce `%assign` a `%store`. Instance interního objektového modelu, která je výsledkem interpretace je poté přivedena na výstup podmínkového funkčního uzlu, který povolí zápis do registru vždy, když daný objekt nabývá požadované hodnoty.

Pokud je výsledných podmínkových uzlů více (podmíněných přechodů mezi bloky bylo více), jsou všechny jejich výstupní vodiče svedeny do funkčního uzlu provádějící operaci logického `and`. Výstup tohoto uzlu je přiveden na povolovací vstup registru. Operace `and`, kterou uzel provádí, zajistí, že zápis do registru bude povolen pouze v okamžiku, kdy se výstupní hodnoty všech podmínkových uzlů budou nacházet v logické „1“.

5.3.3 Analýza inicializačních hodnot registrů a pamětí

Posledním úkonem, který je při interpretaci instrukcí třeba provést, je vytvoření inicializačních objektů, které udávají počáteční hodnoty registrů. Inicializace registru na počáteční hodnotu v jazyce Verilog nebo VHDL je po převodu do VVP formátu v instrukční části popsána následující konstrukcí:

```
T_0 ;
     %pushi/vec4 0, 0, 11;
     %store/vec4 v0x879f5e0_0, 0, 11;
     %end;
     .thread T_0;
```

Příklad 5.3: Zdrojový kód VVP inicializující hodnotu registru.

Jak vidíme, vlákno začínající na návěští `T_0` se nepohybuje v nekonečné smyčce. Toto vlákno pouze jednou provede svůj instrukční kód – tedy přiřazení hodnoty 0 do registru daného operandem – a poté je ukončeno instrukcí `%end`. Taková to inicializační vlákna je nutné rozpoznat a pro přiřazení inicializační hodnoty do registru negenerovat reprezentaci v rámci interního modelu. Namísto toho je vytvořena instance třídy `Init`, která popisuje, jaká hodnota je na začátku simulace přiřazena do kterého registru.

Kompletní popis modelu uvedeného v příkladech v kapitolách 2 a 3 v jazycích VHDL, Verilog a VVP včetně jejich konverze do formátu VAM je uveden v příloze A.

5.4 Optimalizace interního modelu

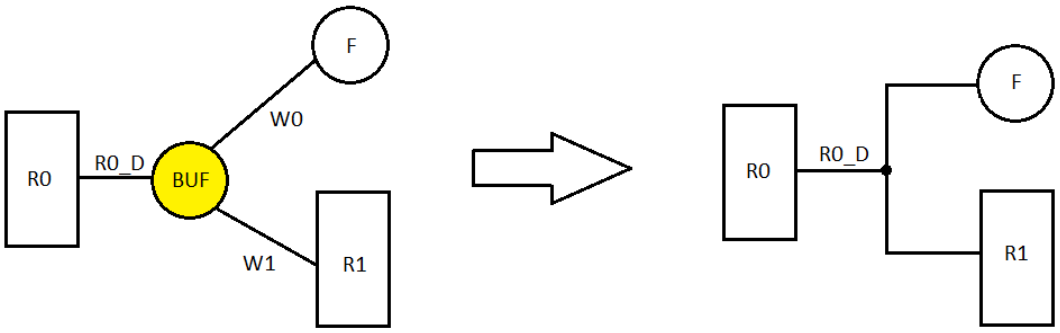
Optimalizací interního objektového modelu, reprezentujícího hardwarový obvod, existuje celá řada. Pro příklad můžeme uvést:

- odstranění vodičů, které nejsou připojeny na žádné komponenty,
- sloučení všech výstupních vodičů registrů a funkčních uzlů do jednoho vodiče, který je rozveden na všechny výstupy,
- odstranění propagačních funkčních uzlů.

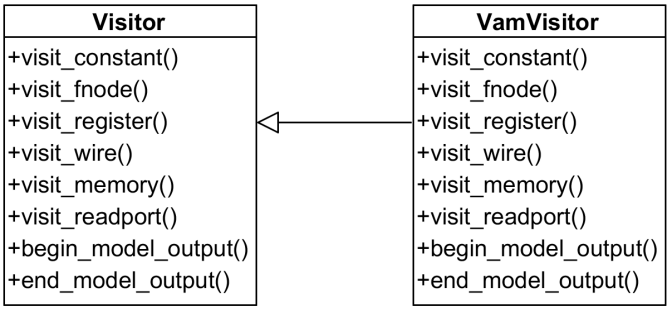
Výsledkem optimalizací může být zjednodušení a zpřehlednění modelu. Některé optimalizace také mohou zrychlit další zpracování modelu a jeho simulaci.

Optimalizace interního modelu navržená v rámci této bakalářské práce spočívá v eliminaci propagačních uzlů vytvářených při generování VVP reprezentace. Tyto uzly ve VVP slouží k propagaci konstant a také ke spojení prvků, které nemohou být spojeny přímo. V interní objektové reprezentaci překladače jsou konstanty reprezentovány samostatnými objekty a všechny elementy jsou propojeny vodiči typu `Wire`. Proto propagační uzly již neplní žádnou funkci a mohou být optimalizovány.

Při odstraňování těchto uzlů prochází překladač celý interní model, a pokud narazí na propagační funkční uzel, smaže jej, i všechny vodiče, které pro tento uzel fungují jako



Obrázek 5.7: Ukázka optimalizace propagačního uzlu.



Obrázek 5.8: Třídní diagram použitého návrhového vzoru Visitor.

výstupy. Poté je pro všechny objekty, do kterých byly výstupní vodiče uzlu připojeny jako vstupy, přiveden jako vstup vstupní vodič optimalizovaného uzlu. Celý proces je znázorněn na obrázku 5.7.

5.5 Generování cílové reprezentace

Pro generování cílové reprezentace je použit návrhový vzor `Visitor`[11]. Třídní diagram navržené třídy `VamVisitor` je ilustrován na obrázku 5.8. Použití tohoto návrhového vzoru umožňuje obecný návrh interního modelu, a jeho případný převod do jiné koncové reprezentace přidáním další specifické třídy implementující rozhraní třídy `Visitor`.

Toto rozhraní deklaruje virtuální metody pro návštěvu každé třídy interního objektového modelu, které jsou poté definovány v odvozené třídě generující specifický cílový kód. Každá třída interního modelu implementuje metodu `accept`, která jako parametr vyžaduje instanci třídy `Visitor`. Z metody `accept` se volá metoda návštěvníka `visit` pro danou instanci interního modelu, která má být převedena do cílové reprezentace.

Kapitola 6

Implementace překladače

Navržený překladač byl realizován podle návrhu aplikace popsaného v předchozí kapitole. V této části textu se zaměříme na některé implementační detaily.

6.1 Zvolený implementační jazyk, nástroje a převzaté zdrojové kódy

Pro implementaci překladače byl zvolen jazyk C++ z důvodu možnosti realizace objektového návrhu a také použití nástrojů `Flex`[19] a `Bison`[13]. Tyto nástroje umožňují snadné generování lexikálního a syntaktického analyzátoru.

Lexikální analyzátor je vytvořen ze zdrojového souboru obsahujícího seznam pravidel pro rozpoznání jednotlivých výrazů analyzovaného jazyka (terminálů). Tato pravidla jsou zapsána formou regulárních výrazů.

Syntaktický analyzátor je vygenerován nástrojem `Bison`. Zpracovávaný zdrojový soubor obsahuje gramatiku popisující strukturu vstupních dat zpracovávaných analyzátorem. Pro pravidla gramatiky jsou definovány části kódu, které se provádějí ve chvíli rozpoznání daného pravidla. V implementovaném překladači jsou z těchto úseků analyzátoru volány metody třídy `VvpParser`, vytvářející příslušné instance VVP modelu.

Pro potřeby implementovaného překladače byly z projektu Icarus Verilog převzaty zdrojové soubory pro vygenerování lexikálního a syntaktického analyzátoru formátu VVP. Soubor `parse.y` obsahuje specifikaci bezkontextové gramatiky v tzv. Backus-Naurově formě. Do pravidel zmíněné gramatiky byly v rámci této bakalářské práce doplněny sémantické akce prováděné překladačem pro jednotlivá pravidla. Pomocí vytvořených sémantických operací je při chodu překladače generován VVP model. Díky těmto převzatým souborům je překladač schopen zpracovávat výstup nástroje Icarus Verilog a také ověřovat správnost těchto vstupních dat. Převzetí těchto zdrojových kódů značně usnadnilo analýzu VVP a realizaci celého kompilátoru.

6.2 Spuštění programu

Program je spouštěn v příkazové řádce. V této části technické zprávy jsou popsány parametry spuštění a jejich funkce. Program podporuje jak krátké, tak i dlouhé parametry implementované podle standardních konvencí.

- `-h / --help` vypíše nápovědu programu, poté je program ihned ukončen,

- `--input / -i file.vvp` specifikuje vstupní soubor formátu VVP, který bude překladačem zpracován, povinný parametr při každém spuštění, kromě výpisu nápovědy,
- `--output / -o file.vam` udává jméno výstupního souboru pro model v jazyce VAM, tento parametr je nepovinný, pokud není zadán, je VAM reprezentace vypsána na standardní výstup,
- `--init / -n file.ini` specifikuje jméno souboru, do kterého se vypíše inicializační informace pro vygenerovaný VAM model, pokud parametr není zadán, jsou tato data vypsána na standardní výstup,
- `--no-opt` parametr vypne optimalizace interního modelu,
- `--mute` vypne výpisy na standardní výstup informující o průběhu překladu.

6.3 Tok řízení programu

Průběh programu je řízen funkcí `main()` ze zdrojového souboru `main.cpp`. Po spuštění jsou nejdříve zpracovány parametry programu pomocí knihovní funkce `getopt_long()`. Tato funkce podle zadných parametrů nastaví řídící proměnné programu a cesty k jednotlivým souborům, se kterými program pracuje.

Následuje vytvoření instance třídy `VvpParser` u níž je invokována metoda `parse()`. Ta pomocí lexikálního a syntaktického analyzátoru zpracuje vstupní VVP soubor a vytvoří odpovídající VVP model. V rámci VVP modelu nejsou objekty propojeny ukazateli, jednotlivé instance pouze obsahují řetězce reprezentující unikátní identifikátory souvisejících objektů. Následuje analýza instrukcí, které jsou rozděleny do základních bloků potřebných při jejich interpretaci.

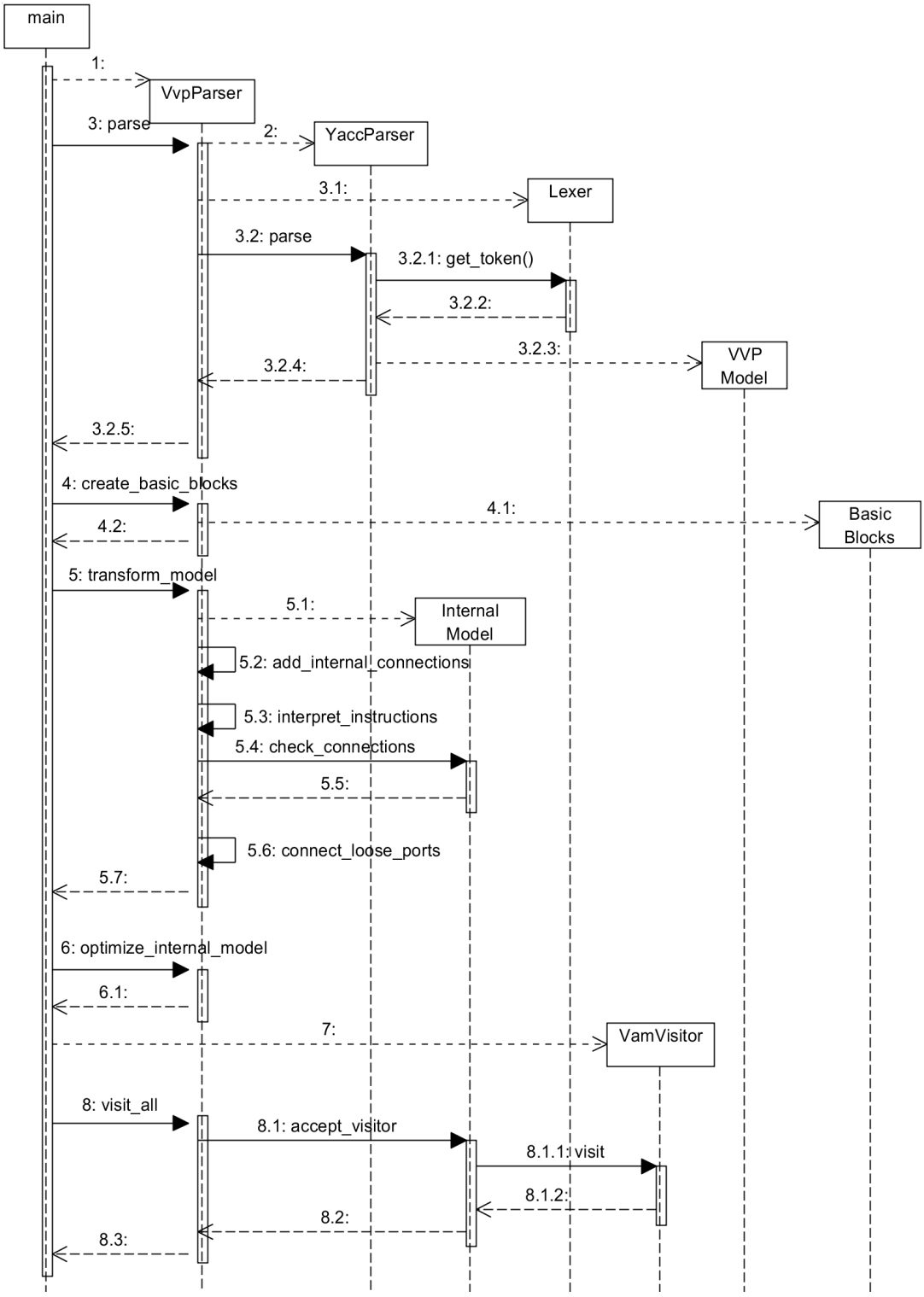
Poté je z funkce `main()` invokována metoda překladače `transform_model()`. V rámci této metody je pro každou instanci VVP modelu vytvořena odpovídající instance interního objektového modelu podle diagramu transformace modelu znázorněného na obrázku 5.2. Pomocí metody `add_internal_connections()` jsou instance interního objektového modelu propojeny ukazateli, které nahrazují reference na související objekty implementované ve VVP modelu pomocí řetězců. Dále je provedena interpretace instrukční části vstupního VVP souboru pomocí metody `interpret_instructions()`. Tato funkce zavede do interního objektového modelu propojení datových vstupů registrů a pamětí s příslušnými vstupními objekty a na jejich povolovací vstupy přivede výstupní vodiče odpovídajících podmínkových funkčních uzlů.

Po interpretaci instrukcí je u každé instance interního objektového modelu invokována metoda `check_connections()`. Pomocí této metody je provedena kontrola, zda jsou všechny objekty propojeny pomocí instancí třídy `Wire`. Pokud jsou například dvě instance třídy `Fnode` propojeny ukazateli přímo mezi sebou, je tato reference odstraněna a nahrazena nově vytvořeným vodičem, kterému je jako vstup přiřazen první funkční uzel a jako výstup druhý funkční uzel.

Na konec transformace modelu je na všechny vstupní porty registrů a pamětí, na které nejsou připojeny žádné vodiče, přiveden pomocí metody `connect_loose_ports()` vodič ve stavu vysoké impedance. Tato operace usnadňuje generování cílové reprezentace, protože v modelu popsaném jazykem VAM musí být na vstupní porty registrů a pamětí připojen vodič.

Poté jsou volitelně provedeny optimalizace interního modelu. Implementovaná optimalizace spočívá v eliminaci propagačních uzlů, vzniklých při vytváření VVP souboru nástrojem Icarus Verilog. Tyto propagační funkční uzly, označované identifikátory `BUFZ` a `BUFT`, jsou vyhledány v interním objektovém modelu a odstraněny. Jejich vstupní a výstupní objekty jsou poté propojeny tak, aby funkce modelovaného obvodu zůstala ekvivalentní. Celý proces optimalizace probíhá podle návrhu popsaného v kapitole 5.

Nakonec je ve funkci `main()` vytvořena instance třídy `VamVisitor`, která je předána metodě `visit_all`. Tato metoda zajistí generování VAM reprezentace modelu do příslušného výstupního souboru. Poté jsou dealokovány všechny zdroje využívané překladačem a program je ukončen.



Obrázek 6.1: Tok řízení při běhu překladače.

Kapitola 7

Testování překladače

Testování je nedílnou součástí vývoje software a musí být důkladně provedeno před nasazením aplikace do provozu. Z hlediska implementace programu pracujícího v příkazové řádce je nejdůležitější oblastí testování ověření funkčnosti aplikace. V této kapitole jsou popsány testy použité při vývoji navrženého překladače a platformy, na nichž byl překladač testován.

7.1 Navržené testy

Použitá testovací sada ověřující funkčnost překladače se skládá ze dvou hlavních druhů testů:

- tzv. *unit* testy,
- integrační testy.

Unit testy jsou použity v první fázi testování a zaměřují se na nejmenší testovatelné části aplikace. Obvykle jde o testy jednotlivých částí aplikace na úrovni modulů a tříd. Unit testy navržené pro implementovaný překladač jsou zaměřeny na správné vytvoření jednotlivých instancí tříd VVP modelu podle vstupního souboru ve formátu VVP. Tyto testy dále ověřují, že každý z objektů VVP modelu je při transformaci převeden na odpovídající instanci interního objektového modelu se správně nastavenými atributy.

Pokročilejší unit testy ověřují funkčnost propojení jednotlivých komponent interního objektového modelu pomocí ukazatelů a také bezchybnou eliminaci propagačního uzlu při optimalizaci zkoumaného modelu.

Integrační testy jsou ve srovnání s unit testy složitější. Zaměřují se na komunikaci jednotlivých modulů aplikace a na fungovaní aplikace jako celku. Při testování překladače byly použity integrační testy zaměřené na následující oblasti:

- Zapojení jednoduchého obvodu z registru a funkčního uzlu.
- Optimalizace propagačních uzlů v jednoduchém obvodu.
- Přivedení správného vodiče na povolovací vstup registru.
- Zapojení více vodičů na datový vstup jednoho registru.
- Interpretace instrukcí pro datový vstup registru a paměti.

- Interpretace instrukcí pro povolovací vstup registru a paměti.
- Čtení dat z paměti v rámci instrukční části VVP.
- Zpracování celého procesoru – `tinycpu.vhd`.

Pomocí výše popsaných testů byla ověřena funkčnost navrženého a implementovaného překladače.

7.2 Testované platformy a verze použitého software

Překladač implementovaný v rámci této bakalářské práce byl zprovozněn a testován na následujících platformách.

Operační systém Ubuntu 14.04 s programovým vybavením:

- g++ 4.8.2,
- flex 2.5.35,
- GNU bison 3.0.2,
- Icarus Verilog,
- vhd2vl 2.4.

Na operačním systému Windows lze implementovaný překladač nainstalovat pomocí následujících nástrojů:

- mingw32 – g++ 4.8.2,
- win_flex 2.5.37,
- win_bison 2.7,
- Icarus Verilog,
- vhd2vl 2.4,
- Msys2 – mingw-w64 (pro instalaci Icarus Verilog).

Kapitola 8

Závěr

Cílem této bakalářské práce bylo navrhnout a implementovat překladač jazyků pro popis hardware VHDL a Verilog do grafové reprezentace v jazyce VAM. Vyvíjený parser má sloužit pro potřeby formální verifikace obvodů popsaných právě jazyky Verilog a VHDL. Díky možnosti převodu návrhů hardwarových obvodů z vysokoúrovňových návrhových jazyků do grafové reprezentace jsou pro analýzu a verifikaci těchto obvodů dále použitelné verifikační nástroje vyvíjené v rámci projektu HADES na FIT VUT Brno.

Funkčnost překladače byla otestována jak základními testy ověřujícími funkčnost překladu jednotlivých komponent hardware, tak i na jednoduchých procesorových jádrech dostupných na serveru OpenCores [18] (např. `tinycpu.vhd`). Testování probíhalo také s využitím aplikace Dfsim, vyvíjené v rámci jiné bakalářské práce panem Davidem Kovaříkem [16] s plánovaným odevzdáním v roce 2015. Dfsim umožňuje interaktivní simulaci obvodů popsaných v jazyce VAM. Obě aplikace jsou použitelné při procesu verifikace hardware společně, protože formát výstupních dat implementovaného překladače je uzpůsoben pro zpracování simulátorem Dfsim.

Další rozšíření funkčnosti aplikace se může ubírat dvěma směry. Prvním z nich je přidání podpory zpracování dalších hardwarových elementů formátu VVP a rozšíření množiny interpretovaných instrukcí. Tímto postupným vylepšováním překladače by se jistě dalo dosáhnout zpracování větších a komplexnějších obvodů.

Druhou možností rozšíření možností kompilátoru je převod interního modelu do jiné koncové reprezentace. Tato činnost by neměla být náročná vzhledem k dostatečné obecnosti interního objektové modelu a použití návrhového vzoru `Visitor`. Toto vylepšení by spočívalo v návrhu a implementaci nové třídy, která by definovala metody deklarované v rozhraní abstraktní třídy `Visitor`.

Literatura

[1] Aldec, Inc.: Nástroj Aldec Active-HDL [online]. `https://www.aldec.com/en/products/fpga_simulation/active-hdl`, 2015 [cit. 2015-02-05].

[2] Algolist: Popis prohledávacího algoritmu DFS [online]. `http://www.algolist.net/Algorithms/Graph/Undirected/Depth-first_search`, 2015 [cit. 2015-04-19].

[3] Bartsch, G.: VHDL parser Zamiacad [online]. `http://sourceforge.net/p/zamiacad/code/ci/master/tree/`, 2015 [cit. 2015-02-05].

[4] Biere, A.: The AIGER And-Inverter Graph (AIG) Format [online]. `http://fmv.jku.at/aiger/FORMAT.aiger`, 2007 [cit. 2015-04-28].

[5] Biere, A.: AIGER 1.9 And Beyond [online]. `http://fmv.jku.at/hwmcc11/beyond1.pdf`, 2011 [cit. 2015-02-05].

[6] Brayton, R. K.; Mishchenko, A.: ABC: An Academic Industrial-Strength Verification Tool. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, 2010, s. 24–40.

[7] Charvát, L.; Smrčka, A.; Vojnar, T.: Hades – Nástroj pro verifikaci hardware [online]. `http://www.fit.vutbr.cz/research/groups/verifit/tools/hades/`, 2015 [cit. 2015-02-05].

[8] Cimmati, A.; Clarke, E.: Nástroj NuSMV - symbolic model checking [online]. `http://nusmv.fbk.eu/NuSMV/papers/sttt_j/html/node2.html`, 2015 [cit. 2015-04-17].

[9] Doolittle, L.: Překladač jazyka VHDL do jazyka Verilog [online]. `http://doolittle.icarus.com/~larry/vhd2vl/`, 2010 [cit. 2015-04-30].

[10] FIT VUT Brno: Výzkumná skupina VeriFIT [online]. `http://www.fit.vutbr.cz/research/groups/verifit/`, 2015 [cit. 2015-02-05].

[11] Gamma, E.; Helm, R.; Johnson, R.; aj.: *Design patterns: elements of reusable object-oriented software*. 1995, ISBN:0-201-63361-2.

[12] Ghost, K. L.: Miscellaneous EDA Utilities [online]. `http://www.edautils.com/`, 2015 [cit. 2015-02-05].

[13] GNU OS: Webové stránky nástroje Bison - generování syntaktického analyzátoru [online]. http://www.gnu.org/software/bison/, 2014 [cit. 2015-04-20].

[14] Johannes Kepler Universität Linz: Knihovna a nástroje AIGER [online]. http://fmv.jku.at/aiger/#intro, 2015 [cit. 2015-02-05].

[15] Kölbl, A.; Pixley, C.: Constructing Efficient Formal Models from High-Level Descriptions Using Symbolic Simulation. *International Journal of Parallel Programming*, ročník 33, č. 6, 2005: s. 645–666.

[16] Kovařík, D.: *Interaktivní simulátor pro grafy toku dat.* Bakalářská práce, FIT VUT v Brně, plánované odevzdání 2015.

[17] Mentor Graphics: Nástroj ModelSIM [online]. http://www.mentor.com/products/fv/modelsim/, 2015 [cit. 2015-02-05].

[18] OpenCores: Webové stránky OpenCores [online]. http://opencores.com/, 2015 [cit. 2015-04-20].

[19] Paxson, V.: Webové stránky nástroje Flex - generování lexikálního analyzátoru [online]. http://ftp.gnu.org/old-gnu/Manuals/flex-2.5.4/html_node/flex_19.html, 1990 [cit. 2015-04-20].

[20] Prudel, S.: VHDL parser Jindent [online]. http://home.wtal.de/software-solutions/vhdl-parser/, 2015 [cit. 2015-02-05].

[21] Reintjes, P. B.: VHDL parser v jazyce SWI-Prolog [online]. http://cecs.wright.edu/~tkprasad/VHDL-AMS/README.html, 2002 [cit. 2015-02-05].

[22] Smrčka, A.: Přednáška č. 3 - Data Flow [online]. https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/ITS-IT/lectures/3-dataflow.pdf, 2015 [cit. 2015-04-19].

[23] Verific Design Automation: Společnost Verific [online]. http://www.verific.com/, 2015 [cit. 2015-02-05].

[24] Williams, S.: Simulátor IcarusVerilog [online]. http://iverilog.icarus.com/, 2014 [cit. 2015-02-05].

[25] Williams, S.: Popis souborů formátu VVP [online]. https://github.com/steveicarus/iverilog/blob/master/vvp/README.txt, 2014 [cit. 2015-04-18].

[26] Williams, S.: Popis instrukční sady a simulátoru formátu VVP [online]. https://github.com/steveicarus/iverilog/blob/master/vvp/opcodes.txt, 2015 [cit. 2015-04-18].

[27] Xilinx: Nástroje Xilinx Vivado [online]. http://www.xilinx.com/products/design-tools/vivado.html, 2015 [cit. 2015-02-05].

Příloha A

Kompletní příklady zdrojových kódů

Jazyk VHDL

```
entity InvReg  is
   port( CLK : in std_logic );
end InvReg;


architecture arch of InvReg is
  signal R0_WE  : std_logic;

  signal R0     : std_logic_vector(7 downto 0);
  signal R0_Q   : std_logic_vector(7 downto 0);
  signal R0_D   : std_logic_vector(7 downto 0);
begin

   R0_Q <= R0;
   R0_D <= not R0_Q;

    process(CLK)
    begin
        if ( rising_edge(CLK) ) then
           R0_WE <= '1';

           if(R0_WE = '1') then
              R0 <= R0_D;
           end if;
        end if;
    end process;

end arch;
```

Jazyk Verilog

```
module main;

   reg   CLK;
   reg   R0_WE;

   reg[7:0]   R0;
   wire[7:0]  R0_D;
   wire[7:0]  R0_Q;

   always @( posedge CLK ) begin

      R0_WE <= 1;

      if((R0_WE == 1'b 1)) begin
         R0 <= R0_D;
      end

   end

   assign R0_Q = R0;

   assign R0_D =~ R0_Q;

endmodule
```

Formát VVP

```
#! /usr/local/bin/vvp
:ivl_version "0.10.0 (devel)" "(s20150105)";
:ivl_delay_selection "TYPICAL";
:vpi_time_precision + 0;
:vpi_module "system";
:vpi_module "vhdl_sys";
:vpi_module "v2005_math";
S_0x8e4e028 .scope module, "main" "main" 2 17;
 .timescale 0 0;
L_0x8e6ea50 .functor NOT 8, v0x8e6e7c0_0, C4<00000000>,
                           C4<00000000>, C4<00000000>;
v0x8e4e800_0 .var "CLK", 0 0;
v0x8e6e7c0_0 .var "R0", 7 0;
v0x8e6e838_0 .net "R0_D", 7 0, L_0x8e6ea50;  1 drivers
v0x8e6e8b8_0 .net "R0_Q", 7 0, v0x8e6e7c0_0;  1 drivers
v0x8e6e930_0 .var "R0_WE", 0 0;
E_0x8e4de88 .event posedge, v0x8e4e800_0;
    .scope S_0x8e4e028;
T_0 ;
```

```
    %wait E_0x8e4de88;
    %pushi/vec4 1, 0, 1;
    %assign/vec4 v0x8e6e930_0, 0;
    %load/vec4 v0x8e6e930_0;
    %cmpi/e 1, 0, 1;
    %jmp/0xz  T_0.0, 4;
    %load/vec4 v0x8e6e838_0;
    %assign/vec4 v0x8e6e7c0_0, 0;
T_0.0 ;
    %jmp T_0;
    .thread T_0;
# The file index is used to find the file name in the following table.
:file_names 3;
    "N/A";
    "<interactive>";
    "inv-bp.v";
```

Jazyk VAM

```
(model inv_bp

(sig W_0000 1 (vt unknown) (meta (name CLK_out)))
(sig W_0001 8 (vt unknown) (meta (name R0_out)))
(sig W_0002 1 (vt unknown) (meta (name R0_WE_out)))
(sig W_0003 1 (vt unknown) (meta (name C_0000_out)))
(sig W_0004 1 (vt unknown) (meta (name L_0000_out)))
(sig W_0005 1 (vt unknown) (meta (name C_0001_out)))
(sig v0x8e6e838_0 8 (vt unknown) (meta (name R0_D)))
(sig v0x8e6e8b8_0 8 (vt unknown) (meta (name R0_Q)))

(fnode C_0000 (input ) (output W_0003)
       (assign (:= W_0003 1))(meta (comment "constant") (value 1)))
(fnode C_0001 (input ) (output W_0005)
       (assign (:= W_0005 0))(meta (comment "constant") (value 0)))

(reg v0x8e4e800_0 1 (d W_0005) (q W_0000 )
       (we W_0005) (meta (name CLK)) )
(reg v0x8e6e7c0_0 8 (d v0x8e6e838_0) (q W_0001 v0x8e6e8b8_0 )
       (we W_0004) (meta (name R0)) )
(reg v0x8e6e930_0 1 (d W_0003) (q W_0002 )
       (we W_0003) (meta (name R0_WE)) )

(fnode L_0000 (input W_0002) (output W_0004)
       (assign (:= W_0004 (? (== 1 W_0002) 1 0))))
(fnode L_0x8e6ea50 (input W_0001) (output v0x8e6e838_0)
       (assign (:= v0x8e6e838_0 (  W_0001))) )

)
```

Příloha B

Obsah CD

CD přiložené k bakalářské práci obsahuje tyto soubory a složky:

- `README.txt` – popis instalace a spuštění aplikace pro platformy Unix a Windows,
- `src` – složka obsahující zdrojové kódy překladače, `makefile` pro instalaci programu a skripty `vhdl2vam.sh` a `vl2vam.sh` pro spuštění celého řetězce nástrojů při převodu VHDL / Verilog souborů do VAM reprezentace na OS Unix a skripty `vhdl2vam.bat` a `vl2vam.bat` zajišťující stejnou činnost na OS Windows.
- `install` – složka s nástroji potřebnými pro běh překladače (iverilog, vhd2vl, flex, bison),
- `examples` – složka obsahující soubory se zdrojovými kódy v jazycích VAM, VHDL, Verilog a VVP,
- `doc` – text bakalářské práce ve formátu PDF.